Chapter 12
Scrum Team Structures

Scrum teams are essential assets of a Scrum organization. How they are structured and related to one another can significantly affect an organization's success with Scrum. In this chapter I discuss different ways to structure Scrum teams. I begin by discussing the distinction between a feature team and a component team. I then focus on the issue of coordinating multiple, collaborating Scrum teams.

Overview

If you have one small product, you don't need to worry much about the content of this chapter. Just create one cross-functional development team using the characteristics I described in Chapter 11, and make sure to properly fill the ScrumMaster and product owner roles. From a Scrum team perspective, you're all set to go!

Let's say, however, that your one, cross-functional Scrum team becomes a high-performance engine for delivering business value and your organization starts to grow. Or, you are already a larger organization and after developing your first product with Scrum, your use of Scrum begins to spread. In both instances you might soon find yourself needing to coordinate the work of multiple Scrum teams whose combined effort is required to deliver increasingly greater business value.

How should you structure these teams so that they are high performing and well coordinated? I address this question by considering whether you should create feature or component teams and what approaches can be used for coordinating multi-team activities.

Feature Teams versus Component Teams

A **feature team** is a cross-functional and cross-component team that can pull end-customer features from the product backlog and complete them. A component team, on the other hand, focuses on the development of a component or subsystem that can be used to create only part of an end-customer feature.

In Chapter 6 I discussed how a GPS manufacturer might create a routing component team to manage the sophisticated code associated with determining a route from an origin to a destination. Any time there is a request for new GPS features that involves routing, the routing-specific pieces of those features would be assigned to the routing component team for development.

Component teams are sometimes referred to as asset or subsystem teams. Often a community of practice made up of people with a similar specialty skill (see Figure 13.4) also functions like a component team. On these teams, all members likely report to the same functional manager and might operate as a shared, centralized resource to other teams. One example might be the centralized UX department that creates UI designs for other teams.

Scrum favors feature teams. Unfortunately, many organizations prefer component teams, often because they believe that a team of experts who are trusted to make safe and effective changes to a particular area of code should own that area of the code. Their thinking is that people unfamiliar with the code could inadvertently break it in unpredictable ways. They prefer having a component team responsible for developing that code and making changes on behalf of others.

Let's say we are developing a product whose features frequently cut across three component areas (see Figure 12.1).

In this example, there is no feature team that works on a complete product backlog item. Instead, a feature is selected from the top of the product backlog and is split into its component-level pieces (the three pieces shown inside the dashed rectangle of Figure 12.1). This splitting is done either collectively by the members of the component Scrum teams, or perhaps by an architect.

The individual pieces of the feature are then placed into the respective product backlogs of component teams (for example, the first piece is put into the component area 1 product backlog—"CA 1 PB" in the figure). Each component team performs

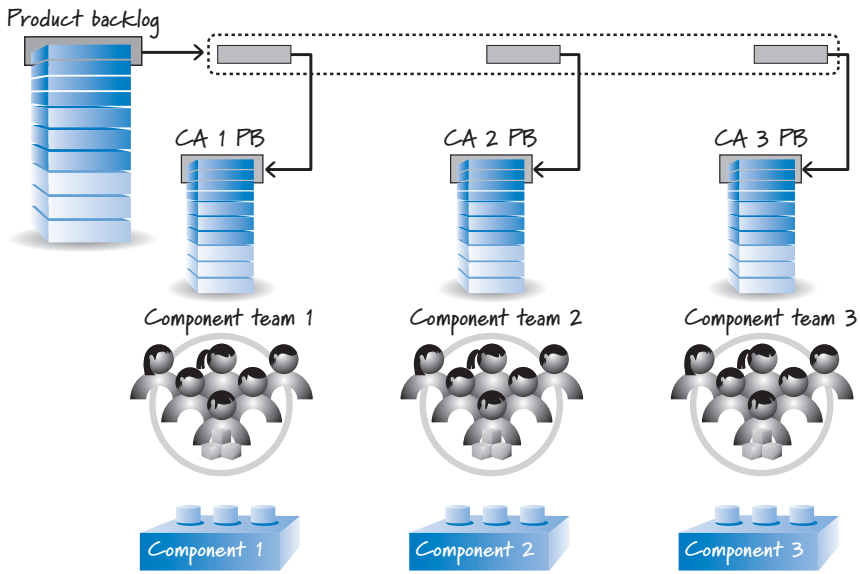

FIGURE 12.1 One product and multiple component teams

Scrum against its own component-area-specific backlog, completing component-specific pieces of end-customer features but not the full feature. Using a technique like scrum of scrums, which I will discuss shortly, the component teams integrate their individual component-level pieces back together and deliver the full end-customer feature.

If there is only one product channeling requests to the component teams, this approach will probably work. However, most organizations frequently form component teams around component areas that they intend to reuse on multiple products. Figure 12.2 shows how the work might flow if there were two products channeling requests to the same component teams.

Each feature-level product backlog contains end-customer-valuable items that can span the multiple component areas. So, in Figure 12.2, there are now two products that need component teams to work on their specific component-level pieces.

Imagine you are the product owner of one of the component teams. You now have to prioritize competing requests from two products, while at the same time coordinating with the other component-level teams to make sure the various pieces get integrated together at the appropriate time.

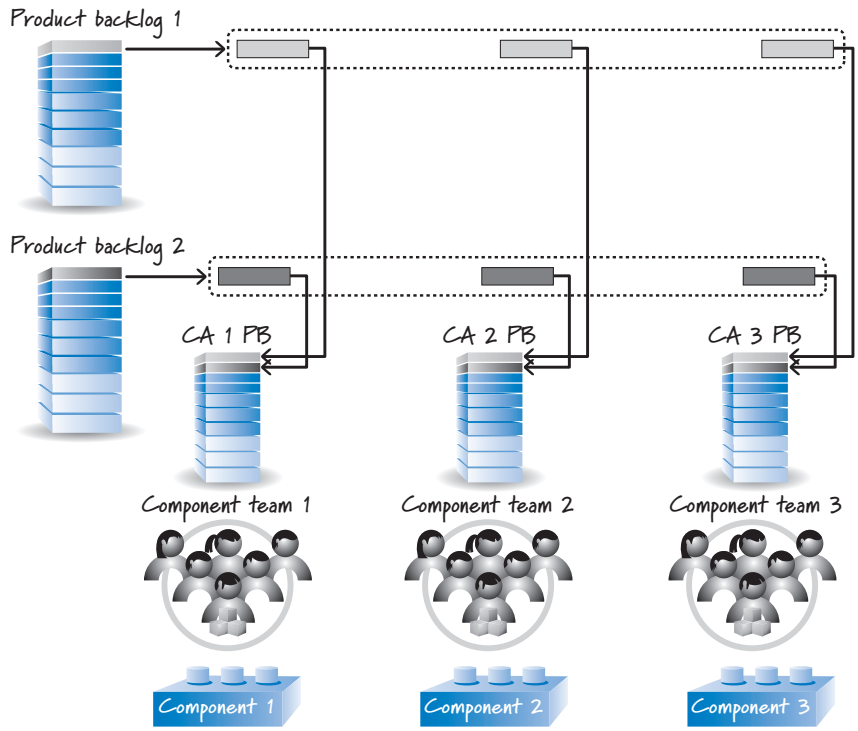

FIGURE 12.2 Two products and multiple component teams

With two products the logistics of this problem are probably still manageable. However, what if the organization works on 10 or 15 products at the same time, and each of those products is dropping component-level pieces into the component team backlogs? At this scale the logistics of figuring out the proper order to work on the individual pieces within a particular component team backlog, while at the same time coordinating and integrating with all of the other component teams, become unmanageable.

In my experience, most organizations using component teams recognize that there's a problem when things begin to fall on the floor (the baton drops, causing a break in value-delivery flow). It usually goes something like this. A senior manager asks a feature-level product owner, "How come the customer feature isn't ready?" The response: "Well, all but one of the component teams finished the pieces we assigned to them. Because that last team didn't finish, the feature isn't done." The manager might then say, "Why didn't that team finish the piece you gave them?" The response might be "I asked, and I was told that they had 15 other competing requests for changes in their component area, and for technical reasons they felt it made more sense to work on the requests from other products before ours. But they still promise to finish our piece—perhaps in the next sprint."

This is no way to operate a business. We can never be certain when (or even if) we can deliver a feature—because the responsibility for delivery has been distributed among two or more component teams, each of which might have very different priorities. Using component teams this way multiplicatively increases the probability that a feature won't get finished, because there are now multiple points of failure (each component team) instead of one (a single feature team).

Is there a solution to this problem? Well, a very good solution would be to create cross-functional feature teams that have all of the skills necessary to work on multiple end-customer features and get them done—without having to farm out pieces to component teams. But what about the principal reason that most organizations create component teams—having a single trusted team to work in a component area? Won't feature teams lead to chaotic development and maintenance of reusable components with large amounts of technical debt? Not if we have well-formed feature teams that, over time, share code ownership and collectively become trusted custodians of the code.

A transitory approach en route to this multi-feature-team model with full shared code ownership is to organize the teams as shown in Figure 12.3.

In this approach, the concept of a feature team has been reintroduced. There is now a single feature team that can pull an end-customer-valuable feature off of the product backlog. This feature team has complete responsibility for doing the work and managing the logistics of getting the feature done.

Trusted component teams also remain in this model to help maintain the integrity of the individual component areas. These component teams still have a product backlog that typically contains technically oriented work that needs to take place within the component area (perhaps technical debt repayment work).

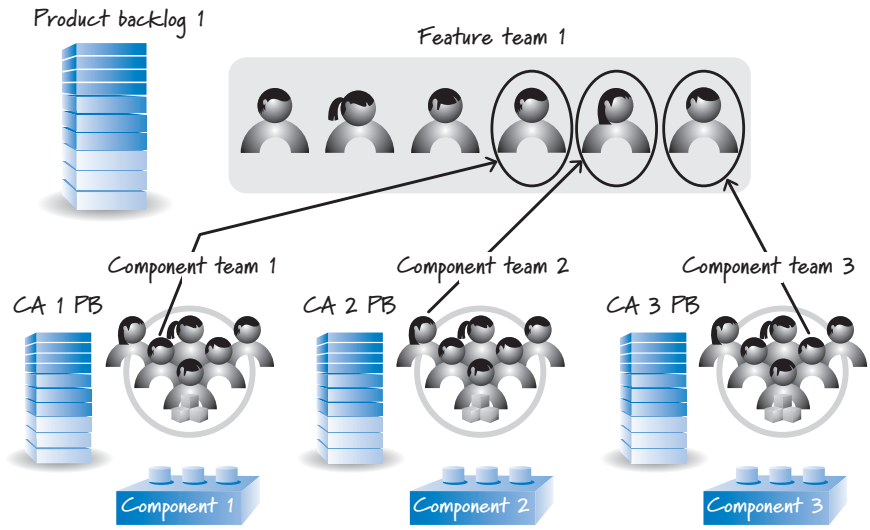

FIGURE 12.3 Combined feature team and component teams

Also, as illustrated in Figure 12.3, a member of a component team can be assigned to be a member of a feature team. This person has the dual responsibility of being both a pollinator and a harvester (Goldberg and Rubin 1995).

In the role of pollinator, the component team members pollinate feature teams with knowledge of the component areas to help better promote shared code ownership within the feature teams. In the role of harvester, component team members collect changes that the feature teams need to make within component areas and discuss those changes with their colleagues on the component teams, each of whom might also be collecting changes to the same component areas. In these discussions, the component team members can better ensure that the component-area changes needed to satisfy the requests of multiple feature teams can be coordinated. Additionally, the people making the component-area changes can do so in a coherent, nonconflicting fashion, thus better ensuring the conceptual integrity of the component areas. The component team members can also keep each other apprised of potential reuse opportunities because everyone has a shared understanding of the changes being harvested in the component areas.

Like the pure component team approach, this approach also can break at large scale—but for different reasons, ones that we can actually address. For example, when I introduced this approach to people at one large company, they remarked, "But our features can cut across up to 50 different systems [components]. We can't move 50 people up to be on one feature team." Although a feature may indeed cut across 50 components, it is rare that all 50 of the components need to directly interact with one another. As a result, we don't need one team of 50 people, but instead we can

create several "feature teams" that form around smaller clusters of components that do have a high degree of interaction (see Chapter 13, Figure 13.5 and Figure 13.6, for examples) and then coordinate the efforts of these teams with the multiple-team techniques that I will describe later in this chapter.

Another way the approach in Figure 12.3 can break is if the organization is working on 40 different products and has only four team members in a component area. It doesn't make sense to assign people out to ten different feature teams at the same time. However, this problem can be solved by reducing the number of products being developed concurrently (see Chapter 16), training (or hiring) more people who have expertise in the component area, and, preferably, better promoting shared code ownership (which is the long-term vision).

In my experience there is no one-size-fits-all solution to the issue of feature versus component teams. Most large and successful Scrum organizations tend to have a blended model composed mostly of feature teams with the occasional component team—when the economics of having the component team as a centralized resource make sense. Sadly, many organizations favor the reverse—mostly component teams with the occasional feature team. These organizations pay a great price in the form of delays from frequently disrupted flow.

Multiple-Team Coordination

Scrum scales not by having increasingly larger development teams, but instead by having multiple right-sized Scrum teams. When there is more than one Scrum team, however, we have the issue of how to coordinate those teams. Two techniques for multiteam coordination are the scrum of scrums and the more comprehensive form of multiteam coordination known as a release train.

Scrum of Scrums

In Chapter 2 I noted that each day during sprint execution the development team performs a daily scrum. Each team's daily scrum includes only the members of that Scrum team.

A common approach to coordinating work among multiple teams is the scrum of scrums or SoS (see Figure 12.4).

This practice allows multiple teams to coordinate their inter-team work. The team that performs the SoS is composed of individual members of the various development teams. Each development team determines which member to send to the scrum of scrums based on who can best speak to the inter-team dependency issues. Although I prefer to have consistency of representation, the person representing a team at the SoS can change over time based on who is best able to represent the team and speak to the issues at that point in time.

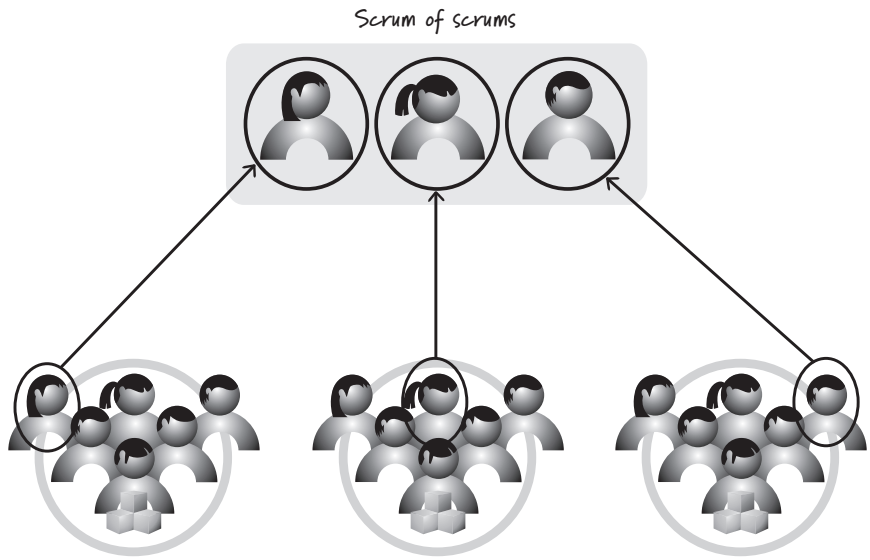

FIGURE 12.4 Scrum of scrums

Some teams send both a development team member and their ScrumMaster (who might be shared among two or several Scrum teams) to the SoS—collectively being cautious of not allowing the overall number of participants to become too large. It might even make sense to have a ScrumMaster at the level of the scrum of scrums. If such a role exists, it could be filled by one of the individual team ScrumMasters or by a ScrumMaster not working directly with any of those teams.

There are multiple approaches to conducting a scrum of scrums, and the participants should decide which approach works best for them. Typical of all approaches, the SoS is not held every day but instead a few times a week as needed. Participants at the scrum of scrums answer similar questions to the ones answered at the daily scrum:

- What has my team done since we last met that could affect other teams?
- What will my team do before we meet again that could affect other teams?
- What problems is my team having that it could use help from other teams to resolve?

Some teams timebox their scrum of scrums to be no more than 15 minutes, just like an individual Scrum team's daily scrum. And they defer problem solving to occur after the scrum of scrums has completed so that only those participants whose involvement is necessary for problem resolution need attend.

An alternative approach is to extend the scrum of scrums beyond the 15-minute timebox. Although the participants might begin each SoS with a 15-minute timebox for answering the three questions, the SoS continues past that 15-minute activity, providing the participants the opportunity to problem-solve issues that came up.

In theory, the scrum of scrums can be scaled to multiple levels. Let's say there is a product being developed with many teams. Typically these teams would group together into feature-area clusters. Within a given cluster of teams a traditional SoS can be used to help coordinate the work of a feature area. It would also make sense to have a higher-level SoS, called a scrum of scrum of scrums (more easily thought of and pronounced as a "program-level scrum"!) that would help coordinate the work among the clusters. Although this approach can work, there are other techniques for coordinating large numbers of teams. One in particular is the release train, which I will discuss next.

Release Train

A **release train** is an approach to aligning the vision, planning, and interdependencies of many teams by providing cross-team **synchronization** based on a common cadence. A release train focuses on fast, flexible flow at the level of a larger product.

The train metaphor is used to imply that there is a published schedule of when features will "leave the station." All of the teams participating in the development of the product need to get their cargo onto the train at the appointed time. As in any country with reliable trains, the release train always departs on time and waits for no one. Likewise, if a team misses the train, it need not fret because there will be another train departing at a known time in the future.

Leffingwell defines the rules of a release train as follows (Leffingwell 2011):

- Frequent, periodic planning and release (or potentially shippable increment—PSI) dates for the solution are fixed (dates are fixed, quality is fixed, scope is variable).
- Teams apply common iteration lengths.
- Intermediate, global, objective milestones are established.
- Continuous system integration is implemented at the top, or system, level, as well as at the feature and component levels.
- Release increments (PSIs) are available at regular (typically 60- to 120-day) intervals for customer preview, internal review, and system-level QA.
- System-level hardening iterations are (or may be) used to reduce technical debt and to provide time for specialty release-level validation and testing.
- For teams to build on top of similar constructs, certain infrastructure components—interfaces, system development kits, common installs and licensing utilities, user-experience frameworks, data and web services, and the like—must typically track ahead.

Figure 12.5 shows a partial release train picture based on Leffingwell's definition.

The release train is a rich concept with multiple levels of detail, including portfolio and release levels. As I mentioned in Chapter 6, the release train is based on an enterprise backlog model that contains three levels of backlogs: portfolio backlog (with epics owned by portfolio management), program backlog (with features owned by program management), and the team backlogs (with sprintable user stories owned by product owners). Figure 12.5 illustrates only the team level. Details of portfolio-level planning and release-level planning will be discussed in Chapter 16 and Chapter 18 respectively.

The team-level train in Figure 12.5 shows a total of nine teams clustered into three feature areas. Each team within a feature area performs its own sprint, drawing work from its associated feature-area backlog. Using a technique like scrum of scrums, all the teams within a feature area coordinate and integrate their work each sprint.

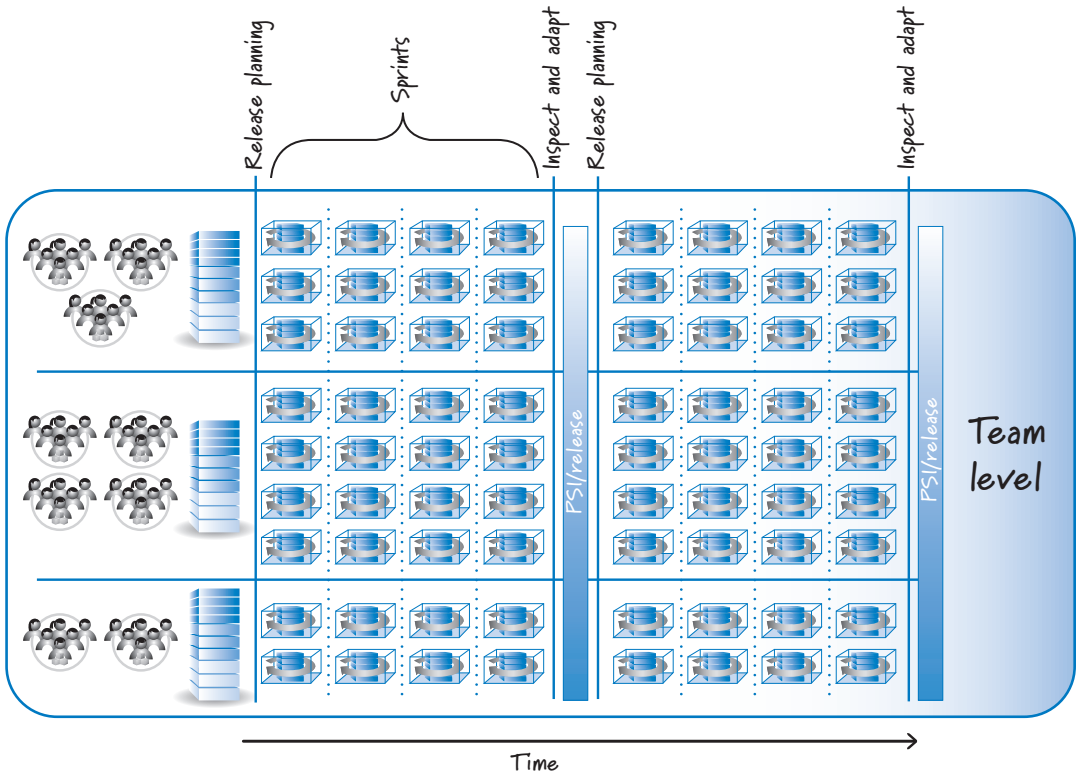

FIGURE 12.5 Release train structure

Also, as often as is practical, there should be system-wide integration and testing across the feature areas. Some teams reserve the last sprint before the train departs to be a time for *hardening* what has been developed in the previous sprints and integrating and testing the results across the various feature areas (for example, sprint 4 in Figure 12.5 might be a hardening sprint). As team skills mature, the need for a hardening sprint should diminish.

The durations of all sprints for teams participating in the release train are identical, and all sprints are aligned. The result is that sprints of every team start and end on the same dates. Doing this enables **synchronization** to take place not only within a given feature area but across all teams working on the product.

Finally, a PSI (release increment) is available after a fixed number of sprints, which in the case of Figure 12.5 is four sprints. Knowing that release points will occur at reliable times allows the organization to synchronize its other activities to known future dates. At these release points, the organization can choose to deploy a PSI to its customers (if that is the business-appropriate thing to do) or use it instead to confirm that the work performed within the individual feature areas has been integrated and tested across the feature areas, and to solicit internal review.

Each release train begins with a release-planning meeting that spans all of the teams working on the PSI (see Figure 12.5). That means that potentially hundreds of people are simultaneously participating in a joint planning event. I have to admit it is fascinating to see this happen. Here is an overview of planning at this scale.

First, you will need a big room! The chief product owner (see Figure 9.13) leads this activity and typically kicks things off. Individual Scrum team members colocate at the same table or area of the room (preferably near open wall space where they can hang up their artifacts). Scrum teams in the same feature area cluster nearby. Once the chief product owner provides the big picture for the PSI, teams gather with other feature-area teams. Feature-area product owners then provide the big-picture overview of their feature areas for the upcoming release train.

Individual Scrum teams then begin mapping out their sprints by slotting features into specific sprints. This activity is referred to as sprint mapping, which I will discuss in Chapter 18. Because Scrum teams are actually working on the creation of a larger, multiteam deliverable, there will be inter-team dependencies. To help manage these dependencies, at any point a member of one Scrum team can get up and walk over to another Scrum team (perhaps carrying a note card or Post-It) and ask the other Scrum team if it can complete the piece of work identified on the card during the upcoming release train. If it can, the team making the request can then commit to the dependent feature.

During all of this, people with multiteam responsibilities, such as the chief product owner, feature-area product owners, and shared architects, can circulate from table to table to help ensure that the big picture is understood and that a coherent overall plan for the upcoming release train makes sense. Of course, a Scrum team can always request that one of these shared individuals come over to assist.

Once the release train sprints are completed and we arrive at a PSI release point (train departure), we then perform release-train-level inspect-and-adapt activities. The first is a PSI review of the full set of cargo that was placed on the release train. The second is a retrospective at the release train level that is focused on how to make future release trains more efficient. Then we are off to the release planning for the next release train.

Closing

In this chapter I discussed different ways to structure Scrum teams. I began by describing feature teams that are cross-functionally diverse and sufficient to pull an end-customer feature from a product backlog and get it done. I then compared feature teams with component teams that work in specific component, asset, or architectural areas, which represent only parts of what needs to be integrated into end-customer features. I went on to show a blended model of feature and component teams and how it could be used to help an organization transition to a point where it has mostly feature teams, each with excellent shared code ownership.

I then discussed how to coordinate multiple Scrum teams, starting first with a traditional Scrum practice called the scrum of scrums and then describing a concept called a release train, which can be used to coordinate the activities of a large number of Scrum teams. In the next chapter I will depart from the traditional Scrum team roles and discuss the role of managers in a Scrum organization.